Extending the Do! Framework with Dynamic Collections

Pascale Launay, Jean-Louis Pazat

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Extending the Do! Framework with Dynamic Collections

Pascale Launay and Jean-Louis Pazat

No˚3757

Septembre 1999

THÈME 1

ISSN 0249-6399

Extending the Do! Framework with Dynamic Collections

Pascale Launay* and Jean-Louis Pazat†

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n° 3757 — Septembre 1999 — 14 pages

Abstract: The aim of the *Do!* project is to ease the task of programming distributed applications using Java. We provide a shared-memory parallel programming model, possibly including additional informations about object locations; we generate distributed programs from parallel programs by using the user's object location specifications, and by automatic transformation of user-defined classes. Writing a distributed application is a three-step process: in the first step, the programmer writes his application as a shared-memory parallel program, by defining the components of the program; in the second step, the programmer defines the component locations on distinct hosts; in the third step, the *Do!* preprocessor transforms the shared-memory parallel program into a distributed-memory program. In this paper, we present an extension to our computing framework in order to create and activate tasks at runtime; this extension allows the introduction of load balancing strategies in the framework by taking into account runtime informations in the component location descriptions. This extension has been integrated in the parallel and distributed frameworks, without any modification in the *Do!* preprocessor.

Key-words: Java, framework, parallel programming, distribution, dynamic creation of tasks

(Résumé : tsvp)

* Pascale.Launay@irisa.fr
† Jean-Louis.Pazat@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Une extension du framework Do! par l'introduction de collections dynamiques

Résumé : L'objectif du projet *Do!* est de simplifier la programmation d'applications distribuées en Java. Le modèle de programmation est un modèle parallèle centralisé, avec la possibilité de donner des indications pour le placement des objets ; un programme parallèle est transformé en un programme distribué à partir des spécifications de placement des objets, et par transformation automatique de classes définies par le programmeur. L'écriture d'une application distribuée se fait en trois étapes : dans la première étape, le programmeur écrit son application comme un programme parallèle centralisé, en définissant les composants de son application ; dans la deuxième étape, le programmeur définit le placement de ces composants sur les différentes machines ; dans la troisième étape, le préprocesseur *Do!* transforme le programme parallèle centralisé en programme distribué. Dans cet article, nous présentons une extension de notre framework pour la création et l'activation de tâches à l'exécution ; cette extension permet l'introduction de stratégies d'équilibrage de charge dans le framework en introduisant des informations dynamiques dans la description du placement des objets. Cette extension a été intégrée dans les frameworks parallèle et distribué, sans modification du préprocesseur *Do!*.

Mots-clé : Java, framework, programmation parallèle, distribution, création dynamique de tâches

1 Introduction

The aim of the *Do!* project [14, 15] is to ease the task of programming distributed applications using Java. The *Do!* environment provides a shared-memory parallel programming model, freeing the programmer from the task of locating and accessing objects mapped on distinct hosts and implementing communications, using for example the Socket class or the Java Remote Method Invocation (RMI) package [10]. The first version of the *Do!* environment provides a static programming model. In this paper, we present an extension to our computing framework in order to create and activate tasks at runtime; this extension allows the introduction of load balancing strategies in the framework by taking into account runtime informations in the component location descriptions. In section 2, we give an overview of the *Do!* environment. The extension for dynamic creation of tasks is presented in section 3 and we show how distribution is managed to exploit dynamic information in section 4. In section 5, we present some approaches in related projects.

2 Overview of the Do! environment

The *Do!* environment provides a shared-memory parallel programming model, possibly including additional informations about object locations, and generates distributed programs from parallel programs using the user's object location specifications by automatic transformations of user-defined classes. The *Do!* environment (figure 1) is made up of:

- a parallel framework to express the parallel programming model,
- a distributed framework to express the distributed execution model,
- a preprocessor to transform parallel programs into distributed programs.

The frameworks are composed of standard Java classes so that writing parallel or distributed programs using the *Do!* environment does not require to learn any language extension nor to use a modified JVM. We can use standard Java environments providing the standard Java facilities for multithreading, RMI and reflection.

2.1 Writing a distributed application using Do!

In a first step, the programmer writes an application as a shared-memory parallel program without bothering with the component locations. He writes the components of his program: tasks and data objects. Then the programmer can choose between different task execution models by selecting a class in the *Do!* framework library. The *Do!* parallel framework manages the cooperation scheme between user components.

In a second step, the programmer gives the component locations on distinct hosts by choosing appropriate *layout managers*. Layout managers are part of the framework; they contain the description of the object distribution policy. This allows us to automatically

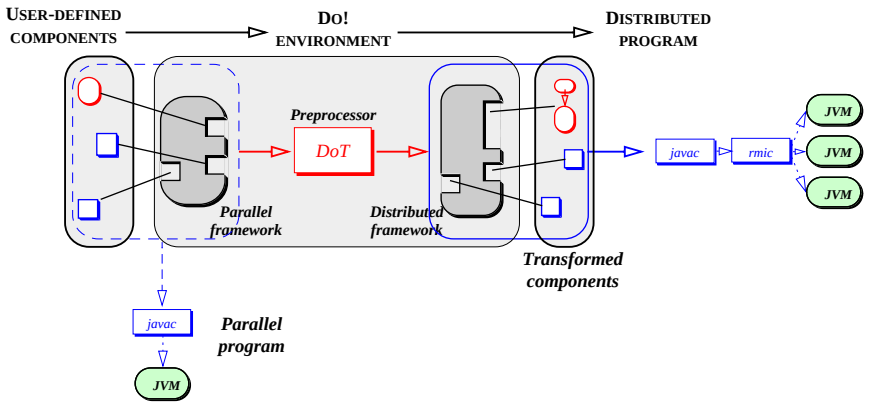

Figure 1: The Do! environment

generate the distributed program from the parallel program. In the distributed framework, layout managers are used to implement the object distribution policy.

In a third step, the *DoT* preprocessor transforms the user's shared-memory parallel program into a distributed-memory program. Because a program is composed of framework classes and user defined classes, program transformations consist in changing the framework used in the program (the parallel framework is replaced by the distributed framework), and transforming components (user-defined classes) in order to allow transparent locations of objects at run-time (mapping and remote accesses).

2.2 Parallel and distributed frameworks

2.2.1 Underlying concepts:

the parallel and distributed frameworks are both based on the concepts of active objects (tasks) used to introduce parallelism, collections to structure parallelism, and layout managers to describe the mapping of objects on a distributed architecture. The parallel and distributed frameworks have the same programming interface, despite the fact that the parallel framework uses *standard* collections, whereas the distributed framework uses *distributed* collections.

Active objects (tasks) have their own activity: when using active objects, several control flows run concurrently through different objects, leading to *inter-object* concurrency; when active objects access other objects of the program, they can communicate (through shared objects), leading to *intra-object* concurrency when concurrent control flows run into a single shared object.

Collections are structured sets of objects such as lists, arrays, trees... Collections are used in the parallel and distributed frameworks to store both passive and active objects

and thus to express structured parallelism. Elements of distributed collections are objects mapped on distinct hosts. Their distribution policy devolves on a special object called a *layout manager*

Layout managers are objects included in the parallel and distributed frameworks to describe the collection distribution: the programmer indicates how to distribute the program components (tasks and data) by choosing a layout manager for each collection. Layout managers have no effect in a parallel program, but are used in the distributed framework: they are responsible for retrieving the processor owning an element from a key identifying the element, and for translating global keys into local keys.

2.2.2 Parallel framework:

parallelism is encapsulated in task collections, that store active objects. A first traversal of a task collection asynchronously activates each element leading to intra-collection parallelism. A second traversal is processed to make a synchronization for tasks termination. The *operator design pattern* [12] provides a modular way to define operations applied to collection elements, independently from the collections and iterators implementations[1].

We have defined a parallel construct, the `Par` class, that relies on two specific operators, the `Start` class (to activate tasks) and the `Join` class (to synchronize with task terminations). From a task collection (storing active objects) and a data collection (storing passive objects), the parallel construct implements the parallel activation of tasks with the corresponding data items, and the synchronization with task terminations, using the `Start` and `Join` operators. Using different types of collections and iterators, we implement different models of parallelism as parallel constructs represented by `Par` subclasses. Figure 2 shows two models of parallelism: the `Par` class defines parallel tasks accessing independent objects, while the `SharedPar` subclass implements parallel tasks accessing a single shared data object.

Nested parallelism is introduced by the fact that a `Par` object is an active object, that can be stored in a task collection and activated in parallel with other tasks.

2.2.3 Distributed framework:

the distributed framework has the same structure as the parallel framework: it is represented by a `Par` class, based on `Start` and `Join` operators, that process traversals of task and data collections. The collections used in the distributed framework are distributed collections: their elements are mapped on distinct hosts. An element of a distributed collection is mapped on a processor according to the collection layout manager; this mapping is made at object creation, through remote creation (implemented in the *Do!* runtime). The layout manager gives the processor owning an element from a key identifying the element in the collection. Provided we know the object key in the collection at object creation, we can use the layout manager to decide where to create the object. For indexed collections (arrays, the Java class `Vector`, indexed lists), the key identifying an element in the collection is an index.

[1] iterators [8] are used to define access to collection elements (traversal).

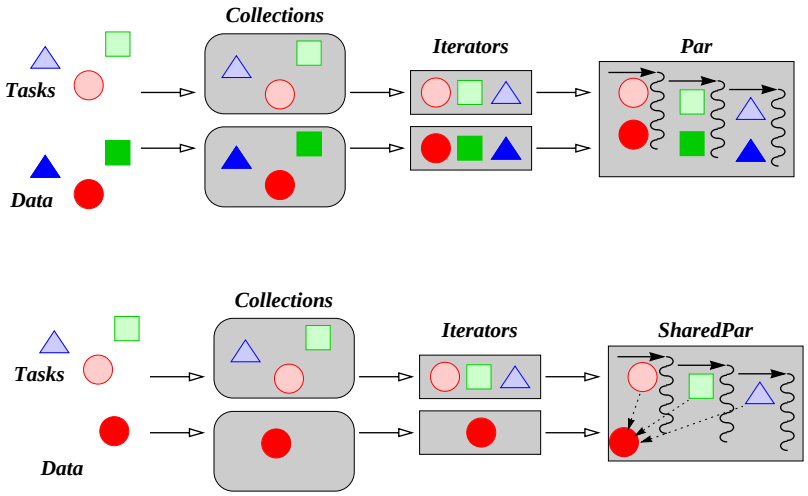

Figure 2: Two models of parallelism: the `Par` and `SharedPar` classes

A specific interface, the `Accessible` interface is used to mark objects that can be accessed from remote hosts (distributed collection elements for example). This interface is used as a "tag" for our preprocessor (section 2.3) to transform components, and contains informations used for dynamic mapping of objects (section 4.2).

Iterators and operators are not distributed: traversals and task activations and synchronizations are processed remotely from a single host; only task execution is distributed. Parallelism exists in the programming model, and distribution does not introduce extra parallelism[2].

2.3 Program transformations

A program is composed of framework classes and user-defined classes (components). To transform of a parallel program into a distributed program we have to change the framework used in the program and transform components.

Framework replacement: the parallel and the distributed frameworks have the same programming interface. The framework used in the program is replaced by only changing the imports in the program.

Component transformations: the user-defined classes must be mapped on distinct hosts and communicate with remote objects. Mapping and communications of user-defined components are not managed by the frameworks. We transform components to allow transparent locations of objects: the mapping of objects on remote hosts, and the communications with

[2] on the contrary, in the parallel operator design pattern [12], the programming model is sequential; data parallelism is introduced through the distribution of collections, iterators and operators.

objects from remote hosts. The only components we transform are those that can be accessed from remote hosts: they implement the `Accessible` interface, defined in the *Do!* framework.

3 Dynamic creation of tasks

The aim of this extension of the *Do!* framework is to allow dynamic creation of tasks in the programming and execution models. In the *static* version, the collections are initialized with active and passive objects, and then the parallel construct is activated synchronously; thus, the collections remain static during task executions. In this *dynamic* version, it is possible to create and insert new elements in collections during task executions using dynamic collections (which size is not fixed, such as lists for example). Tasks inserted dynamically in an active task collection are activated dynamically to run in parallel with other tasks. This extension has been included in the parallel and distributed frameworks, and allows to transform dynamic parallel programs into dynamic distributed programs without any modification to the *Do!* preprocessor. This extension has been implemented in the framework using an event-based model, described in section 3.2.

Layout managers are responsible for the distribution description of collections; we have implemented dynamic layout managers to manage the location of objects inserted in collections dynamically: dynamic layout managers use runtime informations, such as processor load or memory free space, and implement load balancing strategies at task creation, for example. Dynamic management of distribution by layout managers is explained in section 4.

3.1 Concurrency and robustness

Do! frameworks are built using iterators and operators (section 2.2) to implement activations of tasks stored in collections. In a concurrent environment, it is important to ensure that iterators support insertions and removals in collections during a traversal; such iterators are called robust iterators [8]: *"a robust iterator ensures that insertions and removals won't interfere with traversal, and it does it without copying the collection"*. Operators [12] are objects processing elements provided by iterators. By using iterators in the operator design pattern, robustness can be managed both by the iterator and by the operator: in some cases, the iterator can react properly towards an insertion or a removal in the collection, but in other cases the operator has more information to manage the modification correctly.

Especially, if an element that has already been traversed is removed from the collection, the iterator may ignore the modification, but it can be important for the operator to take it into account. As an example, if we define an operator to compute the sum of integers stored in a collection, and if modifications in the collection are allowed during the computation, it is important that the operator takes modifications in the collection into account. If an integer that has already been added is removed from the collection, the operator has to subtract the integer's value before returning.

Moreover, if the operation duration is not limited to the collection traversal, it may be important to react to a modification in the collection even after the end of the traversal. For example, an operator whose purpose is to display a graphical representation of a collection should be able to react properly to a modification in the collection during the whole collection visualization, and modify properly the representation being displayed.

In the *Do!* framework, the operation processed by the `Start` operator is an asynchronous operation, leading to task activations, whose effects end at task terminations. In the extension proposed for dynamic creations of tasks, we allow modifications in the collection during task executions. A task insertion in the collection leads to the task activation if the task collection is active (tasks are still running). We propose here a general solution to manage robustness in a concurrent environment as well as dynamic task activations.

3.2 An event-based solution

The underlying idea is the following: when a modification occurs in a collection, an event is generated and thrown to iterators operating on that collection. An iterator catching such an event can either manage it properly or throw it to the operator using that iterator. If the element touched by the modification has not yet been traversed, the iterator has to manage the event or else the iterator throws the event to the operator.

The operators related to this scheme are the `Start` and `Join` operators, used by the `Par` class. The `Start` operator activates tasks provided by an iterator:

- if an event occurs generated by an insertion in the task collection, the operator activates the new task;
- if the event is generated by a removal from the task collection, the operator stops the task.

In the same way, the `Join` operator has to take into account modifications in the task collection for the final synchronization.

3.3 Asynchronous events and dynamic creation of tasks

The event pattern that we have implemented is based upon the observer design pattern [8], and shown in figure 3. The two main objects are:

- the source (`EventSource`), that generates events,
- the listener (`EventListener`), that reacts to events.

A listener concerned by events generated by a source subscribes to receive event notifications. When an event occurs, the source object creates and activates an event object (`EventObject`); the event object is responsible for notifying all listeners. All informations relative to the event are passed trough a parameter (`EventParam`). Source and listeners are independent: the source does not know his listeners and the event object processing is

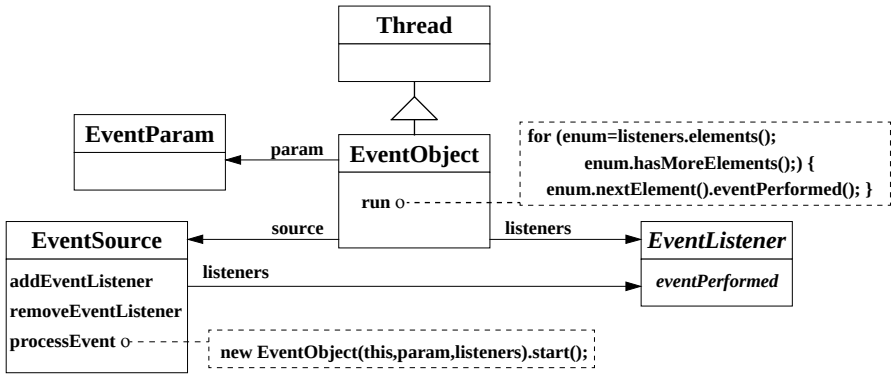

Figure 3: The event infrastructure

asynchronous: notifications are processed in a separate thread, while the source resumes execution.

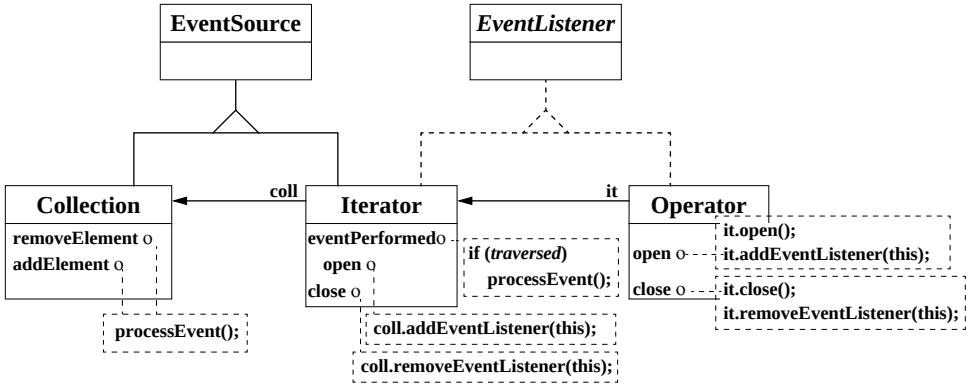

Figure 4: Dynamic iterators and operators

We have included this asynchronous event pattern in the operator design pattern. Collections are event sources and iterators are collection listeners. Additionally, iterators are event sources and operators are iterator listeners. Events are generated by collections when elements are added or removed, and taken into account by iterators and operators. `open` and `close` methods are provided to allow iterator and operator's clients to initiate and stop the event listening process by subscribing or unsubscribing listeners to sources. This pattern is shown in figure 4.

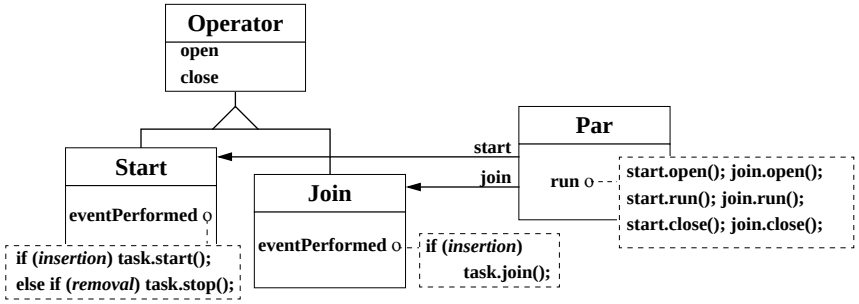

Figure 5: Dynamic creation of tasks

In this framework, insertions and removals in task collections are taken into account during task activation and execution as stated previously (section 3.2). The `Start` and `Join` operators react to events by appropriate actions on tasks being inserted or removed. The `Par` object subscribes operators to listen iterators before processing the collection first traversal (to activate tasks), and operators are unsubscribed after the end of the second traversal (task synchronizations). This is presented in figure 5.

3.4 An example

In this example, we use the ability to add new tasks dynamically with the *Do!* framework. New tasks are created when new machines are getting available. The connection between the runtime (machine insertions) and the program (task creations) is done through asynchronous events.

An application that can be split into several independent computings contributing to the final result can be implemented using a master/servers model: servers run independent computings; master sends queries to servers when needed, collects their contributions and computes the final result. This application computes large prime numbers; its aim is to improve the bounds for the Carmichael conjecture. It is expressed by a master/servers model and requires much computing time (hundreds of hours on one single machine). We run this application on a non-dedicated network of workstations, so we have to be able to react to usage variations in the network to use resources at the best. We map one slave task per available machine, and, in order to use all available resources, when a new machine is added to the *Do!* runtime, we add a new slave task on that machine and integrate it in the computation framework. Using the *Do!* framework, this is done by adding the new task in the task collection. To attain this end, we can use the event pattern again: the program is informed that a new machine has been included in the runtime, and reacts to this event by a new task creation and insertion in the task collection. The runtime is an event source, that generates an event when a new machine is available, and the program is a listener that subscribes to the runtime to be able to react to such events.

4 Dynamic collections and distribution

In the distributed framework, the distribution policy of collections devolves on layout managers. Layout managers have been implemented for static collections (which size is static). We have extended static layout managers for dynamic collections, and defined dynamic layout managers that take into account dynamic informations to compute the mapping of distributed collection elements at runtime.

4.1 Using static layout managers

For arrays, an object index remains the same during the whole object life in the collection; for vectors or indexed lists, an insertion or removal in the collection can modify the indexes of other elements. In the current state of the *Do!* environment, objet migration is not possible; a modification in a collection must not imply a collection redistribution. Therefore, each element is represented by two distinct keys: the first one is used to access elements in the collection; it can be modified by changes in the collection. The second one is used by the layout manager to manage distribution; it is not modified by insertions and removals in the collection.

Dynamic collections are collections which size is unknown statically. For static collections, layout managers can use the collection size to compute a balanced distribution. For dynamic collections, the only way to balance the distribution statically is through a cyclic distribution. So it is possible to manage dynamic collections through static layout managers, but this is not satisfactory, especially for the distribution balancing. Dynamic layout managers are intended to fill in this gap.

4.2 Dynamic layout managers

We have introduced dynamic layout managers to manage efficiently the distribution of dynamic collections; dynamic layout managers compute the collection distribution dynamically, using dynamic informations (runtime parameters for example). Informations used by dynamic layout managers being dynamic, they can change during program execution, and especially between the object creation and the object insertion in the collection. Object location information, computed by the layout manager at object creation, is stored at the object level, through an interface common to all distributed objects: the `Accessible` interface, defined in the *Do!* framework (section 2.3). This location information is used at object insertion in the collection.

The `Dynamic` class, representing a dynamic layout manager has been implemented and included in the *Do!* framework; computing the host owning a new distributed object is made by a `chooseHost` method in the `Dynamic` class. New dynamic layout managers, implementing any dynamic mapping policy, are defined by inheriting the `Dynamic` class and redefining the `chooseHost` method. Thus, we have implemented two dynamic layout managers: one chooses the host having less elements, the other one chooses the host whose processor load is the less (it is currently implemented for Solaris only, using a system call to know the

processor load). In the same way, other more advanced mapping strategies can easily be included in the *Do!* environment.

5 Related work

Many other projects use Java for parallelism and distribution:

- the ProActive PDC [6] project , also known as Java//, is based upon remote and active objects, and a high level synchronization mechanism based on futures. This is implemented using runtime reflection (we use both reflection and compile-time transformations), on top of an open meta-object protocol. This work is based on a programming model described in [5], and previously implemented using Eiffel and C++;
- the JavaParty project [17] is based on language extensions, and distribution is managed by the compiler or the runtime, without any control from the programmer (in the *Do!* project, the programmer controls object distribution). JavaParty allows object migration;
- the High Performance Java project at Indiana University generates parallel programs from sequential Java programs with annotations (JAVAR [4]) and develops a tool to automatically detect and exploit implicit loop parallelism in bytecode (JAVAB [3]). On the contrary, the *Do!* environment relies on a parallel programming model;
- many other projects using Java for parallelism and distribution are presented in [1, 2].

Distributed collections have been developed in the ÉPÉE [11], HPJava [7] and DPJ [9] projects. Those projects provide environments for SPMD parallel programming: the programming model is sequential, and parallelism and distribution is introduced by libraries of distributed static collections (arrays).

The problem of robustness for iterators has been studied in a sequential context for collection libraries in C++ [13, 16]. In the Java Collection Framework (jdk1.2) the problem of concurrent accesses to collections is addressed by using "fail-fast" iterators: concurrent modification of a collection is not allowed during an iteration; a "fail-fast" iterator detecting a modification fails quickly and cleanly by throwing an appropriate exception.

6 Conclusion

We have extended the *Do!* environment to allow dynamic creation of tasks. Modularity of the *Do!* framework allows to add such functionalities and to keep benefits of automatic distribution through program transformations. The first release of the *Do!* environment, available online[3], includes distributed arrays and static layout managers for arrays (`cyclic`,

[3] `http://www.irisa.fr/caps/PROJECTS/Do/`

`block` and `block_cyclic`). Currently distributed indexed lists and static and dynamic layout managers for lists have been implemented.

We intend to improve lists and develop distributed vectors for a second release, that should be available at the time of the final version of this paper. We have developed an application briefly presented in section 3.4; this application and performance measurements will be presented in more details in an incoming paper. A foreseen extension is to include object migration, in order to allow load balancing during task execution.

References

[1] ACM Workshop on Java for Science and Engineering Computation; Simulation and Modelling Program. *Concurrency: Practice and Experience*, June 1997. Las Vegas, Nevada. `http://www.npac.syr.edu/projects/javaforcse/acmprog/prog.html`.

[2] ACM Workshop on Java for High-Performance Network Computing. *Concurrency: Practice and Experience*, February 1998. Palo Alto, California. `http://www.cs.ucsb.edu/conferences/java98/program.html`.

[3] A.J.C. Bik and D.B. Gannon. Exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, 1997.

[4] A.J.C. Bik, J.E. Villacis, and D.B. Gannon. JAVAR. In *ACM Workshop on Java for Science and Engineering Computation; Simulation and Modelling Program*, Jun 1997.

[5] D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[6] D. Caromel and J. Vayssière. A Java framework for seamless sequential, multi-threaded and distributed computing. In *ACM Workshop on Java for High-Performance Network Computing*, February 1998.

[7] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. HPJava : Data parallel extensions to Java. In *ACM Workshop on Java for High-Performance Network Computing*, February 1998.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[9] V. Ivannikov, S. Gaissaryan, M. Domrachev, V. Etch, and N. Shtaltovnaya. DPJ: Java class library for development of data-parallel programs. Institute for System Programming, Russian Academy of Sciences, 1997.

[10] Javasoft. Java Remote Method Invocation specification – revision 1.50. `ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.ps`, October 1998.

[11] J.-M. Jézéquel, F. Guidec, and F. Hamelin. Parallelizing object oriented software through the reuse of parallel components. In *Object-Oriented Systems*, volume 1, pages 149–170, 1994.

[12] J.-M. Jézéquel and J.-L. Pacherie. Parallel operators. In *ECOOP'96*, number 1098 in LNCS, Springer Verlag, pages 384–405, July 1996.

[13] T. Kofler. Robust iterators for ET++. *Structured Programming*, 14(2):62–85, 1993.

[14] P. Launay and J.-L. Pazat. A framework for parallel programming in Java. In *HPCN'98*, number 1401 in LNCS, Springer Verlag, pages 628–637, Amsterdam, April 1998.

[15] P. Launay and J.-L. Pazat. Generation of distributed parallel Java programs. In *Euro-Par'98*, number 1470 in LNCS, Springer Verlag, pages 729–732, Southampton, September 1998.

[16] R.B. Murray. *C++ Strategies and Tactics.* Addison-Wesley Professional Computing Series. Addison-Wesley, New York, N.Y., 1993.

[17] M. Philippsen and M. Zenger. JavaParty – transparent remote objects in Java. In *ACM Workshop on Java for Science and Engineering Computation; Simulation and Modelling Program*, Jun 1997.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399